REMARKS

Pending claims

Assuming entry of this amendment, the pending claims are 1-7 and 10-18, of which claims 1, 12, 14, 17 and 18 are independent.

Allowable Subject Matter

The Examiner indicated that claims 8 and 9 would be allowable if rewritten in independent form to include all the limitations in the claims in their respective chains of dependency. Instead of adding all the "higher up" limitations into claims 8 and 9, which would have created hard-to-follow claims consisting mostly of tracked changes, new independent claims 17 and 18 have been added: Claim 17 includes all the limitations of claims 1, 7 and 8 (with the limitations of claims 7 and 8 after "in which") and claim 18 includes all the limitations of claims 1 and 9 (with the limitations of claim 9 after "in which"). Claims 17 and 18 should therefore be allowable.

Allowed Claims

Original claims 12-16 have been allowed.

Claim Rejections -- 35 USC §112

The Examiner rejected claims 5 and 7, stating that the phrase "substantially uniformly" has no quantifiable meaning.

The paragraph of the application on which these claims are based is paragraph 0114, which reads:

> The term "hash function" is used here merely because using the lower order (or any other set of) bits of P as an index will usually provide a good distribution of indices over the rtc array 230, as is desirable for a hash function. One could also say that h(·) is a "bit extraction" or "mapping" function, or give this function some other name. The purpose would be the same, however. The term "hash function" is used because it is well known in the art and is general. The invention may use any function h of P whose range is preferably the same as (or smaller than) the range of indices of the rtc, that is, [0,m) where m is the length of the rtc array; to minimize the probability of "collisions," that is, two different procedure entry addresses

hashing to the same value, the function h preferably maps P as uniformly as possible over the rtc index range.

First, the applicant observes that even the phrase "substantially uniformly" would be clear and definite enough to skilled programmers in the context of defining a mapping function of the type described. The notion is analogous to the concept of a uniform distribution in probability theory: Just as it is no more likely that a fair die, when rolled, will show any particular number 1-6, the preferred mapping function should be such that it is no more likely that a randomly chosen IL address will be mapped to any particular return target cache (rtc) index than to any other rtc index.

Like rolling dice, however, the distribution will usually not be *perfectly* uniform -- indeed, after the first three rolls, at most only three different numbers can have occurred, so that at least half of the numbers will not have occurred at all. Even over time it will relatively rarely occur that each number will have occurred exactly as many times as all others. The actual outcomes will be *substantially* uniformly distributed, at least over enough runs, even though the distribution mechanism (rolling the fair die) will display a uniform distribution in the mathematical sense.

Well knowing the almost certainty with which future litigators, despite mathematical and even common sense, would try to latch onto and distort the term "uniformly" to mean "perfectly uniformly," the applicant originally wrote "*substantially* uniformly" to conform to mathematical understanding. Nonetheless, claims 5 and 7 have been amended to state that the respective mapping is "with a uniform probability distribution over at least a subset of the return target cache." Note that paragraph 0114 (emphasis added) also supports the "at least a subset" language, as it states that the "range is preferably the same as (***or smaller than***) the range of indices of the rtc." Claims 5 and 7 should now be definite even in the context of claim interpretation, without needlessly implying a restriction to perfect uniformity in the distribution.

Claim Rejections -- 35 USC §102

The Examiner rejected claims 1-11 as being anticipated by the reference "Improving Prediction for Procedure Returns with Return-Address-Stack Repair Mechanisms" by Kevin Skadron ("*Skadron*"). Although the applicant disagrees with the Examiner about the relevance of *Skadron*, as explained below, both the applicant and I wish to thank Examiner Ingberg for making the effort to search and cite non-patent literature, which is all too uncommon in examining computer-related inventions.

Skadron deals with a problem of branch mis-predictions: To increase speed, some processor architectures try to predict along which branch future instruction execution will proceed, then they speculatively fetch and execute the instructions on the predicted branch. One problem with speculative execution in such "pipeline" processor architectures arises when the "guess" is wrong: The *hardware-implemented,* auxiliary return-address stack may become corrupted. In some systems, instructions in more than one branch are speculatively fetched and executed. This ensures that at least one guess is correct, but at the cost of making the problem of return-address stack corruption even worse. This short synopsis of *Skadron's* system is also found in his Abstract. The general problem of mis-prediction leading to return stack corruption is also summarized in one of the other references the Examiner cited, namely, Kaeli ("Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns"), in particular, the left column of page 35.

Key to the *Skadron* system is the return address ***stack***. In particular, *Skadron* assumes a last-in-first-out (LIFO) stack. The nature of such a stack is well known, but is also summarized in section 2.1 ("Basic stack operation") of *Skadron*. Essentially by definition, the nature of such a data structure is that data (in *Skadron's* case, return addresses) are *pushed* onto the top of the stack, whereupon a stack pointer in suitably incremented to indicate the next "top" or "current" position. *Pop* instructions are also associated with a stack, and are essentially just the opposite of a *push* in that the data at the top of the stack is fetched and the stack pointer is decremented (relative to however the "top" is defined") so that data farther down in the stack is next to be popped. If subroutines are nested, then their respective return addresses will be

pushed onto the stack in the order in which the subroutines are called so that the processor can find its way "back" up through the level(s) of called subroutine(s) by popping these return addresses from the stack.

One problem with this arrangement is that a subroutine might, for example, be self-modifying such that it might alter its own return stack, or it might cause an unusual jump, or it might have a "bug," or there may be some other interruption – planned or exceptional – that corrupts the stack in the sense that, when it's time for the processor to return from a subroutine, the correct return address is not at the top of the stack. *Skadron* studies this issue in the context of mis-predictions of speculative execution of one or more future paths in a branching instruction. In other words, *Skadron* deals with a very different problem than the applicant's invention does. These differences of purpose are of course reflected in the significant differences in implementation features.

Given this background, consider now some of the recited elements of the applicant's as yet unallowed independent claim 1:

A) converting a sequence of input language (IL) instructions into a corresponding sequence of output language (OL) instructions

Skadron does not disclose any notion of binary translation. In other words, *Skadron* never converts IL instructions into *any* other form, much less into a specific corresponding sequence of OL instructions. He does not need to. It is not part of his study. The nature of binary translation, and some of its special problems when it comes to subroutine returns, are discussed at length in the specification, starting from about paragraph 0019.

B) executing the OL instructions

Again, *Skadron* has no OL instructions at all, so he also never discloses executing them. *Skadron's* processor therefore does not have to deal with figuring out where to go next in an OL domain at all since he operates solely in a traditional IL domain (no binary translation). *Skadron* can thus more easily confine himself to use of a traditional in-memory general purpose stack augmented by an on-cpu return address stack used for speculative execution.

C) for each call to an IL subroutine made from an IL call site in the IL instruction sequence: i) storing a call site IL return address R_{call} on a stack

The applicant agrees that *Skadron* has this feature, but observes that this is irrelevant: *All* common modern processor architectures have this feature, but this is true regardless of the presence of binary translation.

C) ii) calculating a first ***index*** by evaluating a function with P as an argument, where P is a procedure entry address of the subroutine; iii) storing a corresponding OL return address in a return target ***cache*** at a location indicated by the first index

These two elements are related, and they bring out a key difference between *Skadron* and the applicant's invention: *Skadron* does not calculate a cache (a form of array) index, because he does not need to – the stack pointer is incremented and decremented "automatically," that is, as part of the architected nature of the processor itself. Pointer incrementing and decrementing is fast, but it also means that the position in which a word (for example) of data is placed on the stack is totally independent of the contents of the word itself – it is just pushed onto the top of the LIFO stack, whatever it is. Consequently, if the stack pointer is not pointing to the correct return address when it is time to return from a subroutine, then the processor will simply pop whatever word is next and assume it's the correct return address, even if it isn't even an address at all. According to some known stack-manipulating mechanisms, a stack "unwind" can be used to update the normal instruction stack's pointer, but *not* the special hardware

"mini-stack" (return address stack) in the context of mis-prediction; indeed, it is this very problem that *Skadron* is investigating.

Even if one were to ignore the context with which *Skadron's* study is concerned (mis-predictions in speculative execution) and also ignore that *Skadron* has nothing to do with binary translation (IL instruction(s) converted into corresponding OL instruction(s)) and lacks the features of claim 1 copied above, *Skadron's* stack-bound return mechanism thus suffers from the weakness that the correct return address must be at the top of the stack when it is time for the return.

In contrast, as claimed, the applicant's invention uses a *cache* (array structure) that is accessed using an *index* (not a pointer) that is in turn a *calculated function of the procedure entry address* of the subroutine (not a simple incremented or decremented value wholly unrelated to the entry address). Return addresses therefore do not need to be (and usually will not be) in any special order; in particular, they do not need to be in an order that corresponds to the order in which various subroutines were called. Because the cache index is not just automatically incremented/decremented like a stack pointer, it will never get "out of sync" as long as the procedure entry address is available (which it is) and some other procedure entry address doesn't happen to map to the same cache position (the probability of which can be reduced by increasing the cache size relative to the expected number of addresses to be mapped and adjusting the mapping function accordingly).

C) iv) executing an OL subroutine translation of the called IL subroutine;

Skadron has no OL translation to execute.

D) upon completion of execution of the OL subroutine translation, i) in a launch block of instructions, retrieving an OL target address from the return target cache at the location indicated by a second index; and ii) continuing execution beginning at the OL target address.

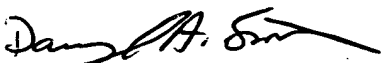
Again, the invention uses indexing into a cache, not operation of a pointer, and it involves execution of translated (OL) instructions. These features are missing from *Skadron*, because he is investigating a problem that arises in the use of a specialized stack, and he is not concerned with binary translation at all.

Conclusion

Claim 1 therefore already has several limitations, deriving from the context of binary translation, that are not only not found in Skadron, but that would make no sense in the context (return address stack contention after branch mis-predictions) in which Skadron is designed to work. Claim 1 should therefore be allowable as filed. Of course, the claims that depend from claim 1 inherit their independent base claim's unique and technically advantageous limitation, as well as adding further limitations of their own. The dependent claims should therefore also be allowable.

Date: February 12, 2007

Respectfully submitted,



VMware, Inc.
3415 Porter Drive
Palo Alto, CA 94304

Darryl Smith
Reg. No. 37,723
Attorney for the Applicant(s)